Opportunistic Linked Data querying through approximate membership metadata

Miel Vander Sande, Ruben Verborgh, Joachim Van Herwegen, Erik Mannens, and Rik Van de Walle

In: International Semantic Web Conference, 92–110, 2015.

Opportunistic Linked Data Querying through Approximate Membership Metadata*

Miel Vander Sande, Ruben Verborgh, Joachim Van Herwegen,
Erik Mannens, and Rik Van de Walle

Multimedia Lab – Ghent University – iMinds
Gaston Crommenlaan 8 bus 201, B-9050 Ledeberg-Ghent, Belgium
miel.vandersande@ugent.be

Abstract. Between URI dereferencing and the SPARQL protocol lies a largely unexplored axis of possible interfaces to Linked Data, each with its own combination of trade-offs. One of these interfaces is Triple Pattern Fragments, which allows clients to execute SPARQL queries against low-cost servers, at the cost of higher bandwidth. Increasing a client's efficiency means lowering the number of requests, which can among others be achieved through additional metadata in responses. We noted that typical SPARQL query evaluations against Triple Pattern Fragments require a significant portion of membership subqueries, which check the presence of a specific triple, rather than a variable pattern. This paper studies the impact of providing approximate membership functions, i.e., Bloom filters and Golomb-coded sets, as extra metadata. In addition to reducing HTTP requests, such functions allow to achieve full result recall earlier when temporarily allowing lower precision. Half of the tested queries from a WatDiv benchmark test set could be executed with up to a third fewer HTTP requests with only marginally higher server cost. Query times, however, did not improve, likely due to slower metadata generation and transfer. This indicates that approximate membership functions can partly improve the client-side query process with minimal impact on the server and its interface.

Keywords: Linked Data, querying, availability, scalability, SPARQL

1 Introduction

For a long period of time, querying Linked Data has been a story of two extremes, with Linked Data documents on the one side and the SPARQL protocol on the other. Currently, neither of them is able to drive real-world applications on the Web. On the one hand, public SPARQL endpoints are limited in number and suffer from frequent downtime [4, 22]. Their resource consumption is hard to predict, caused by the expressiveness of the language and individual user demand. This downtime results in insufficient reliability for client applications. Linked Data documents, on the other hand, are more predictable, but link-traversal-based query methods are significantly slower and result sets have varying levels of completeness, both of which are undesired traits for user applications. The issues with these two query solutions hint at a need for other client/server trade-offs.

* *For Johan De Smedt.* Thanks to Daniel P. Miranker for his suggestions on Bloom filters.

Linked Data Fragments (LDF) [25] aim to analyse such trade-offs by proposing an uniform view on all interfaces to RDF. This reveals a complete spectrum between Linked Data documents and the SPARQL protocol, in which the state-of-the-art of Linked Data publishing can be advanced. This axis can be explored in the following two dimensions.

- **Selector**: allowing different, more complex questions for the server
- **Metadata**: extending the response with more information clients can use

In prior work, Triple Pattern Fragments (TPF) [25] were introduced as an alternative API with low-server cost. This interface offers a single triple pattern as selector and includes an estimated number of total matching triples as metadata. SPARQL queries can be evaluated client-side by combining several TPFs, using the metadata for optimization. Higher query execution time and more bandwidth are accepted in exchange for a small load on the server, thereby striking a more sustainable load balance between client and server. Recently, an algorithm that reduces bandwidth was proposed within the same server restrictions [23]. Another direction for improvement is to have servers support other features along the selector and/or metadata dimensions in addition to TPF.

In this paper, we explore the metadata dimension by adding approximate membership functions (AMF) as a composable feature for Linked Data Fragments APIs. An AMF is a space-efficient data structure that is able to indicate whether a set contains an item. False positives can occur with a fixed probability, but false negatives can not. This work studies their applicability as a server-side feature in addition to TPF, in order to reduce the number of HTTP requests during client-side SPARQL query execution. We study two different AMF techniques: Bloom filters [3] and Golomb-Coded Sets (GCS) [19]. Concretely, we present *i)* an in-depth comparison between different client-side algorithms with or without Bloom and GCS; *ii)* a vocabulary to describe approximate membership functions as metadata for self-descriptive APIs; *iii)* an evaluation of opportunistic querying, where we strive for result completeness first and validate their correctness later.

First, we present the preliminary concepts and related work in Section 2. Then, we discuss the motivation, research questions and hypotheses for this work in Section 3. Next, Section 4 shows how the TPF interface is extended with AMF metadata. After that, we demonstrate how the client benefits from this in Section 5, and how it enables a more opportunistic form of querying in Section 6. Finally, we evaluate the query algorithms with and without AMF metadata in Section 7, and conclude in Section 8.

2 Core concepts and related work

2.1 SPARQL query evaluation using traditional Web APIs

Linked Data can be published on the Web using different APIs, of which data dumps and SPARQL endpoints are highly common [5]. The Linked Data Fragments conceptual framework [25] enables the analysis and comparison of Web APIs by abstracting each API according to how it provides access to parts of a certain dataset. Each such part is called a *Linked Data Fragment* (LDF), which consists of data, metadata, and controls. The *data* is a set of those triples of the dataset that match a given interface-dependent selector. The *metadata* set consists of triples that describe the dataset and/or the current fragment or related fragments. Finally, the *controls* are hypermedia links and/or forms that allow clients to retrieve other fragments of the same or other datasets.

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

Both data dumps and SPARQL endpoint responses can be considered LDFs. A data dump of a dataset employs all triples in that dataset, usually in a compressed archive, as the data. The metadata set contains data such as publication date and/or license. No controls are present, because all available data is contained within the archive. The main drawback of dumps that they cannot be queried "live": they need to be downloaded in their entirety to evaluate queries.

The SPARQL protocol [6] exposes RDF graphs on the Web using the SPARQL query language [10]. Each response to a `CONSTRUCT` or `DESCRIBE` query can be seen as an LDF, where the data consists of the RDF triples in the dataset that match the query. The metadata and control sets are empty; controls are implicitly in the SPARQL protocol. An advantage of SPARQL endpoints is their expressiveness: clients can ask very specific questions about a dataset. However, public SPARQL endpoints suffer from a two-sided availability problem: the majority of datasets is not published as a SPARQL endpoint (543 opposed to 9960 datasets)[1], and endpoints that are on the Web experience frequent downtime [4].

2.2 SPARQL query evaluation using a Triple Pattern Fragments API

In addition to describing existing interfaces, LDF also allows defining new interfaces with different characteristics. The Triple Pattern Fragments (TPF) interface [24, 25] combines the desirable characteristics of data dumps (low server-side cost) and SPARQL endpoints (live queryable). Clients can ask a server for triple patterns; in response, the server sends a TPF, consisting of the triples of the dataset matching the triple pattern (paged to keep the fragment size reasonably small), metadata expressing the total number of matching triples, and controls to retrieve all other TPFs of the same dataset. Complex SPARQL queries are evaluated by clients, which split a query into triple patterns and use the metadata in fragments to determine an efficient execution order. The advantage of TPFs is that they only require low processing power on the server side, and are thus less expensive to host with high availability [25]. The drawback is that SPARQL queries have longer query times than on a SPARQL endpoint. More than 600,000 crawled RDF files are available as TPFs through the LOD Laundromat [21]. DBpedia, arguably the most well-known dataset on the Semantic Web, has an official TPF interface with 99.999% availability [26].

TPFs move the query planning problem to the client. It is up to the client to make optimal use of metadata exposed by the server. The originally proposed query planning algorithm is greedy [25]. Assuming a Basic Graph Pattern (BGP) query, the client downloads results for the triple pattern with the lowest cardinality, based on the count metadata. Possible mappings for each resulting triple are bound to each remaining pattern, of which the one with lowest cardinality is subsequently requested from the server.

Van Herwegen et al. improve the greedy algorithm [23], aiming to minimize the number of HTTP calls by making global instead of local decisions. This is achieved by downloading two triple patterns separately in case this requires fewer HTTP calls. Multiple estimation techniques, based on the intermediate results of the algorithm, are used to predict which query path is least expensive. If the current path is suboptimal, the algorithm continues from the new path. This decrease in HTTP requests results, however, in more computational work for the client because of the more complex join process.

[1] `http://stats.lod2.eu`

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

This paper seeks to provide an optimized balance between server-side cost and query execution time by extending the TPF interface with additional metadata, as we will discuss in Sections 4 to 6. The goal is to maintain a low per-request cost for the server, while reducing the number of requests clients need to execute to evaluate typical queries.

2.3 Approximate membership techniques

In the following, we summarize the Approximate Membership Function (AMF) families of Bloom filters and Golomb-Coded Sets. Both offer approximate membership assessment with a predefined false positive probability, but with different size and speed. Recall and precision are important parameters of an AMF f. Given the set of actual members M and a set of elements T for which we want to test membership, the set of positively tested elements $P_T = \{t \in T : f(t) = \texttt{true}\}$. We define $recall_f(T) = |M \cap P_T|/|M|$ and $precision_f(T) = |M \cap P_T|/|P_T|$. Both Bloom filters and Golomb-Coded Sets have 100% recall, i.e., all valid members of M will always be identified, but less than 100% precision.

Bloom filters A Bloom filter [3] is a bitmap of m bits populated using k different hash functions, initialized with all bits set to 0. An item is added by calculating k locations in the bitmap, which are set to 1. Each one is calculated by using a different hash function to ensure randomness. An item can be tested by calculating k locations using the same hash functions. Hence, both insertion and testing are $O(k)$. The result of a bit-wise `AND` of those locations in the filter determines if the item is a member. If `false`, the item is *definitely* not in the set. If `true`, the item *might* be in the set, because of false positives.

For a desired false positive probability rate p, the bit-size of a Bloom filter is proportional to its number of members n. The required size is $m = -n \cdot \log_2 e \cdot \log_2 p$. For a given m, the optimal number of hashes k that minimizes false positive probability can be calculated with $k = m/n \cdot \ln 2$. Despite their compact representation, their size can be too large for network transfer. A solution is using compressed Bloom filters [15], at the cost of compression and decompression delays.

Golomb-coded sets Golomb-coded sets (GCS) [19] provide a cleaner variation of compressed Bloom filters. The outputs of a single hash function are considered a uniformly distributed list of values instead of a bitmap. The differences between all values form a geometrically distribution with a parameter p. Golomb-coding is applied since it is an optimal encoding for discrete geometric distributions [8].

In terms of size, GCS approaches the theoretical minimum of $m = -n \cdot \log_2 p$ more closely than the equivalent Bloom filter. Compared to compressed Bloom filters, GCS have a minimal size overhead for the same p, but they are more easily chunked and indexed to deal with uncompressed size issues. Compared to plain Bloom filters, the query time is magnitudes slower due to decompression. However, this drawback can be minimized by including an index to quickly find areas of interest in the filter.

2.4 Query evaluation with approximate membership

In the context of RDF querying, approximate membership functions are included in several related works, covering *i)* query routing in networks, *ii)* selectivity estimation for optimizing joins, *iii)* evolutionary querying, and *iv)* local database indexes.

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

Query routing applies Bloom filters in caches and indexes for peer-to-peer, MapReduce or cloud clusters, and Linked Data networks. Most systems [7, 14, 20] construct a *data summary* of neighboring nodes or clusters to make a query forwarding decisions. Some algorithms exchange these filters between nodes to maintain their network [11]. This is common in combination with Distributed Hash Tables (DHT) [11, 27], where a DHT is used for data routing and Bloom filters for efficient communication between nodes.

More directly applicable is *selectivity estimation* of query patterns, e.g., graph patterns, to improve join performance. One approach is to group different chain-patterns, i.e. two distinct triple patterns connected by a single variable, according to their frequency [13]. A Bloom filter tests in what frequency group a chain pattern resides, which optimizes the pattern execution order. Other applications include representing equivalent classes to optimize hash joins, ranges of values for merge joins [16], and distributed n-way joins [2]. Although these works inspire future directions, many require more than a single triple pattern and have high demands for the server. Highly relevant is the proposal to extend the ASK query response [12] with combinations of bindings, i.e. two variables in a triple pattern, to improve source selection in SPARQL query federation frameworks. Bloom filters from different sources indicate overlap and save redundant requests. However, the benefit in a single-server setup is unclear.

Evolutionary querying is an alternative way of SPARQL query processing. Possible solutions are first guessed, and then incrementally refined. Oren et al. use a combination of fingerprinting and Bloom filters to rapidly evaluate approximate answers against large RDF datasets [17]. Although this is a centralized solution, it advocates *anytime* answers, which is in line with the opportunistic querying presented in this paper. The algorithm is initiated with random values, which returns initial results fast, but with low accuracy.

Finally, in the area of databases, Bloom filters are an efficient technique to prevent unnecessary disk access [18]. In such cases, the size of the filter and its impact on transfer delays are not applicable.

3 Problem statement

3.1 Analysis of query execution using Triple Pattern Fragments

The required time for a client evaluate certain SPARQL queries against TPF interfaces can still be unacceptable for responsive applications. A dominant factor in this time is the high number of HTTP requests. Therefore, by analyzing the nature of these requests, we can locate possible areas for changing the client/server trade-offs in the interface. To this end, we executed sample SPARQL queries from the WatDiv benchmark [1] against a TPF interface using the greedy algorithm [25]. WatDiv consists of 20 query templates grouped in four categories, namely linear (L), star (S), snowflake-shaped (F) and complex (C).[2]

The execution logs revealed a high number of requests for triple patterns without variables, i.e. testing the *membership* of a specific triple in the dataset. The templates L2, L4, and F3 respectively produced 50%, 51% and 74% membership subqueries. For S5,

[2] The 20 WatDiv templates are graphically displayed at `http://db.uwaterloo.ca/watdiv/basic-testing.shtml`. Note that the number of templates per category does not necessarily reflect actual query distributions for specific datasets.

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

F5, C1, and C2, this proportion even reached 95% to 98%. Furthermore, the absolute number of requests of some of these templates is high (e.g., F3 needed 1,335 membership subqueries). A third of query templates is thus affected; the remaining 13 templates produced no membership subqueries at all. While these numbers do not allow generalized conclusions, they are certainly an important indication that a reduction of membership subqueries can have a considerable influence on the number of HTTP requests—and thus the overall query execution time.

3.2 Research questions and hypotheses

In the TPF interface, metadata is crucial for clients to evaluate SPARQL queries efficiently. By estimating the total number of matches per triple pattern, patterns with higher selectivity can be followed first [23, 25]. If we augment this metadata, clients might be able to make more informed decisions and hence reduce the number of membership subqueries required to evaluate a SPARQL query, at the cost of higher per-request costs. This paper studies the impact of adding approximate membership functions to fragments in order to reduce the amount of HTTP requests. In this regard, we pose the following research question:

Question 1: To what extent can approximate membership metadata for TPFs reduce the number of HTTP requests necessary to evaluate SPARQL queries?

Probabilistic queries also enable new ways of generating results: uncertain results can be returned early, and validated later on. We investigate this as follows:

Question 2: To what extent can approximate membership metadata for TPFs reduce the time to achieve complete recall of SPARQL query results?

Adding such metadata requires AMFs to be generated on the server side, the impact of which should be investigated:

Question 3: What is the overhead of generating approximate membership metadata on the server CPU load at runtime?

The answers to these questions validate our exploration of the metadata dimension using AMFs. Concretely, we test the following hypotheses about the effectiveness of an interface I', which adds an AMF feature to the baseline TPF interface I. First, given the presence of AMFs, the client should be able to omit a portion of requests over HTTP, hence:

Hypothesis 1: The number of HTTP requests required to evaluate minimum a third of the WatDiv queries against I' can be significantly reduced.

Next, as stated above, the reduction in HTTP requests has a direct impact on the overall execution time, thus:

Hypothesis 2: The time to achieve complete recall when executing WatDiv queries against I' is significantly reduced on average.

Finally, we do not expect much extra load on the server, since an AMF using a non-cryptographic hash function can be computed fast:

Hypothesis 3: The interface I' increases server CPU usage only slightly compared to I for the same queries.

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

4 Extending the TPF interface with AMF metadata

The TPF interface responds with RDF documents and is self-descriptive [25], meaning that *i)* extensions to the TPF interface are features of a composable API, ensuring backward-compatibility; *ii)* clients can discover at runtime which features are supported. Therefore, servers can add an interface feature, e.g., AMFs as extra metadata, without any interference. This section introduces a generic ontology to express membership functions such as AMFs, followed by its implementation as a feature on top of the TPF interface.

We created a membership modeling ontology, which we publish and maintain at `http://semweb.mmlab.be/ns/membership` and denote with the prefix `ms` in the remainder of this paper. It defines `ms:Function` for generic functions and its subclasses `ms:ApproximateMembershipFunction` and `ms:HashFunction`. To allow for Bloom filters and Golomb-coded sets, the former has `ms:BloomFilter` and `ms:GolombCodedSet` as subclasses. Finally, `ms:hashFunction` associates instances of these classes with hash functions that can be instances of algorithms such as `ms:MD5` or `ms:MurmurHash3`.

Using this ontology, we define an interface feature that provides AMF metadata in the metadata graph of responses. In regular TPFs, each fragment contains a `void:triples` statement expressing the approximate total number of triples in the dataset that match the TPF's triple pattern [24]. For instance, each page of the TPF for the pattern "`?x rdf:type foaf:Person`" contains a metadata triple stating there are 96,300 matching triples in the dataset. Given a page size of 100 data triples, these data triples would be spread across 963 pages. Suppose that during the execution of a certain SPARQL query, the client arrives at a list of 215 potential mappings for "`?x rdf:type foaf:Person`". In order to verify with a minimum number of HTTP requests whether these mappings are valid, the 215 TPFs for the corresponding triples need to be downloaded, checking which mappings result in a triple that exists within the dataset.

By defining an interface feature that allows this fragment to contain an AMF, the clients can determine approximately whether a certain `?x` results in a triple of the dataset. Listing 1 shows an example AMF for the triple pattern "`?x rdf:type foaf:Person`". In this case, it is a Bloom filter with two specific Murmur functions as hash functions. The hash functions themselves are not detailed in the listing, but their parameters need to be explicitly available (either in the response or by dereferencing their URL). Listing 1 explicitly specifies that the members of the collection are the triples of the fragment, and that the AMF has been built by using the subject of these triples. This allows the client to interpret how exactly this AMF can be used. For instance, if the triple `dbp:Elvis_Presley rdf:type dbo:Artist` is part of the dataset, then the full URI of `dbp:Elvis_Presley` must yield a positive value in the membership function. Note that the false positive rate is also specified, allowing a client to estimate the certainty of each result. Finally, the AMF data itself has been made available in base64-encoded form.

This metadata allows a client to unambiguously recreate the AMF and verify the approximate membership of elements. Note that this self-descriptive approach does not require a contract between the client and the server, e.g., no hash function has to be agreed upon silently. Furthermore, clients that do not use this metadata feature, such as the original TPF client [25], will not be affected by it and can thus continue to use the interface. It is up to the server's discretion whether or not to provide an AMF on a page. If it is present, an AMF-aware client can use it; if not, the original algorithm without AMFs

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

```
<#metadata> foaf:primaryTopic <#fragment>.
<#metadata> {
    <#fragment> void:triples 96300.         # existing count metadata
    _:membershipFunction a ms:BloomFilter;  # AMF metadata
        ms:hashSize 524288;
        ms:hashFunction <MyMurmur1>, <MyMurmur2>;
        ms:memberCollection [
            ms:sourceCollection <#fragment>;
            ms:projectedProperty rdf:subject
        ];
        ms:falsePositiveRate 0.05;
        ms:falseNegativeRate 0.0;
        ms:binaryRepresentation "QmF...ZTY"^^xsd:base64Binary.
}
```

Listing 1. The self-descriptive AMF metadata in the TPF fragment for `?x rdf:type foaf:Person` allows the client to interpret and evaluate approximate membership.

can be followed. This lets the server choose freely what metadata to include—based on, for instance, the computational effort to create the AMF.

To facilitate implementation, the AMF interface feature is the subject of a specification in the Hydra W3C Community Group, which is available at `http://www.hydra-cg.com/spec/latest/linked-data-fragments/membership-metadata/`.

5 SPARQL query execution with AMF-enabled TPFs

In order to explain the algorithm to query TPFs with AMF metadata, we will consider the following example query for DBpedia:

```
SELECT ?p ?c WHERE {
  ?p a <http://dbpedia.org/ontology/Artist>.                  # tp1
  ?p <http://dbpedia.org/ontology/birthPlace> ?c.             # tp2
  ?c <http://www.w3.org/2000/01/rdf-schema#label> "York"@en.  # tp3
}
```

Query 1. This SPARQL query finds artists born in cities named "York".

Given a regular TPF interface, the algorithms presented in Section 2.2 will compute results for each BGP B by recursively evaluating and binding each triple pattern $tp_i \in B$ in an order determined by the count metadata in their respective fragments. For example, by fetching the first page of the TPFs for Query 1 where $B = \{tp_1, tp_2, tp_3\}$, we obtain the count metadata $\{(tp_1, 96\,300), (tp_2, 625\,811), (tp_3, 2)\}$. Therefore, we start iterating over tp_3, which will supply values for `?c`. This leads to 2 subqueries $B' = \{tp_1, tp'_2\}$ where the remaining triple patterns are bound to concrete values of `?c` (note that tp_1 is unaffected because it does not contain `?c`). For instance, for `?c = dbp:York`, we obtain count metadata $\{(tp_1, 96\,300), (tp'_2, 207)\}$. Query execution thus continues with the smallest fragment tp'_2, which results in 207 subqueries $B'' = \{tp'_1\}$ in which tp_1 is bound

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

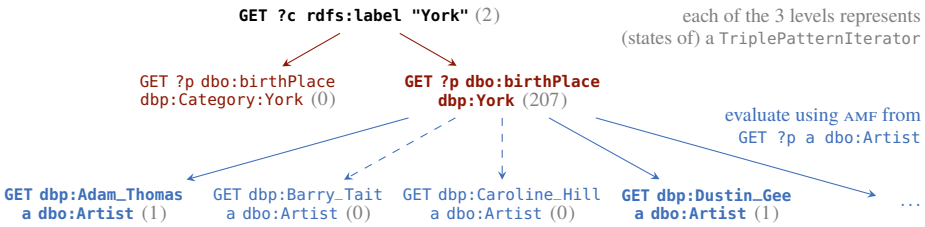

Fig. 1. The triple patterns of Query 1 with the least number of matches at each stage become nodes in the evaluation tree. Note how the third level of consists entirely of membership subqueries (single triples), and can thus be evaluated with the help of an AMF.

to possible values of `?p`. These 207 subqueries are indeed membership queries, because they check the presence of a concrete triple without variables, e.g., "`dbp:Adam_Thomas rdf:type dbo:Artist`". All values of `?p` that result in a match are solution mappings to the query. This process leads to an evaluation tree, as shown in Fig. 1.

An efficient way to realize such evaluation trees are iterator pipelines [9], which allow for *incremental* query results. In existing TPF algorithms [23, 25], two principal iterator types are responsible for SPARQL query evaluation over TPFs: a `TriplePatternIterator` for triple patterns and a `GraphPatternIterator` for BGPs. The whole of Fig. 1 is executed by a `GraphPatternIterator`, which chains together `TriplePatternIterators` for each of the three levels in the tree. Each `TriplePatternIterator` reads solution mappings from the iterator above it and tries to extend them with mappings for a given triple pattern. For instance, the iterator at level 2 with pattern "`?p dbo:birthPlace ?c`" receives mappings for `?c` from the iterator at level 1. For each `?c`, it tries to find mappings for `?p`, which are then passed on to level 3. Finally, the `TriplePatternIterator` on level 3 with pattern "`?p rdf:type dbo:Artist`" either confirms or rejects mappings depending on whether the triple for a given `?p` exists. This produces a total of 207 requests, which amount to 98% of the total HTTP traffic.

Algorithm 1 presents an extension of the original `TriplePatternIterator` [25] to make use of AMF metadata. When a `TriplePatternIterator` is initiated, the corresponding TPF for its initial triple pattern is requested (line 2). This fragment typically already resides in the client cache, since it was formerly requested by a `GraphPatternIterator` for count metadata. If the response contains AMF metadata, a *membership test function* is created and assigned to the iterator (line 4). In our example, this translates to a request for the TPF for "`?p rdf:type dbo:Artist`", which contains an AMF for all mappings of `?p`. If no AMF metadata is found, we assign a constant function *True* that always returns `true` (possible match), so that a verification request is always necessary.

When `GetNext` is called, the `TriplePatternIterator` first reads an upstream mapping μ_s from its source iterator I_s (line 14). Then, we test whether the triple (pattern) tp' resulting from this mapping is present in the current AMF. If the test returns `true`, we have a true positive or false positive, so the TPF corresponding to tp' is fetched and assigned to the iterator. For instance, if the mapping $\{?p = \texttt{Adam\_Thomas}\}$ returns `true`, we retrieve the TPF for "`dbp:Adam_Thomas rdf:type dbo:Artist`" to verify whether this triple is a true or false positive. If the test returns `false`, tp' is a true negative and need not be

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

```
1  Function TriplePatternIterator.Init()
       Data: A source iterator self.I_s; A triple pattern self.tp
2      f_tp ← GET TPF for self.tp;
3      if f_tp contains AMF metadata then
4          self.membership_test ← f_tp.metadata.amf;
5      else
6          self.membership_test ← True where ∀x : True(x) = true;
7      end
8      self.current_fragment ← ∅;
9  end
10 Function TriplePatternIterator.GetNext()
       Output: The next mapping μ_n or nil when no such mappings are left
11     μ ← nil;
12     while μ = nil do
13         while self.current_fragment does not contain unread triples do
14             self.μ_s ← self.I_s.GetNext();
15             return nil if self.μ_s = nil;
16             tp' ← self.μ_s[self.tp];
17             if self.membership_test(tp') = true then
18                 self.current_fragment ← GET TPF for tp';
19             end
20         end
21         t ← an unread data triple from self.current_fragment;
22         μ ← a mapping μ' with dom(μ') = vars(self.tp) and μ'[self.tp] = t;
23     end
24     return μ ∪ self.μ_s;
25 end
```

Algorithm 1. A `TriplePatternIterator` with support for AMF metadata

checked. For instance, if the mapping $\{?p = \texttt{Barry\_Tait}\}$ returns `false`, we are sure the corresponding TPF is empty, so we do not need to perform the HTTP request.

For each negative AMF result, this proposed extension of the algorithm saves an HTTP request. Depending on the type of query, cumulative savings can be extensive, as with Query 1. The positive results, however, still need to be verified in case false positives would have occurred. While we cannot eliminate the verification HTTP calls without endangering the correctness (precision) of query results, it is possible to further reduce the query time, as we will discuss in the next section.

6 Opportunistic query results

In general, query execution does not necessarily end when all valid results have been obtained; it could be that the engine still spends some time to rule out possible result candidates before being able to decide that the result set is in fact complete. Due to the approximate nature of AMFs, it is possible that at a certain point during Algorithm 1, the in-memory result set R already contains all r valid results. However, they cannot be returned yet, because R can still contain a number of false positives f. Only after the

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

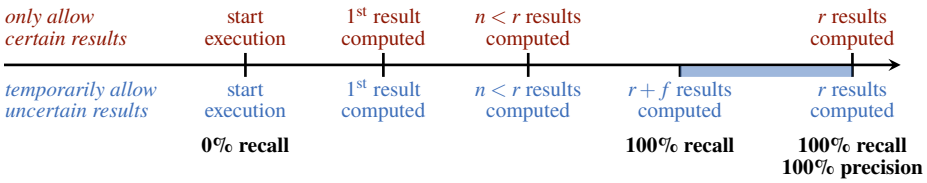

Fig. 2. This SPARQL query execution timeline compares regular and opportunistic query execution, assuming r total query results and f false positives. Note how both approaches achieve 100% recall and precision at a shared point in the end, but there exists a period during which only opportunistic execution reaches 100% recall (shaded).

membership of all positive results of the AMF has been verified against the TPF interface, the f false positives can be discarded and all r matches can be returned safely.

For some use cases, it might be acceptable to *temporarily* consider incorrect results, especially if we are able to indicate which results can be trusted and which results cannot. If at first, we optimistically assume that all positive matches of the AMF are actual matches (i.e., we disregard the false positive rate), the client is able to reach 100% recall earlier, temporarily tolerating a precision below 100%. For each of those approximate matches, the client can express the probability that it is valid, namely $1 - p$ with p the false-positive rate of the AMF. As membership subqueries progress, the client can update the probability for true positives from $1 - p$ to 1, and retract false positives by setting their probability to 0. This opportunistic method of providing query results is important if fast results and eventual full precision are preferred over slower results with immediate precision. At no point in time, incorrect query results are presented as correct results of the query.

Fig. 2 compares regular querying and opportunistic querying. Note in particular how both approaches eventually reach 100% recall and precision *at the same time*. In other words, even though the opportunistic algorithm temporarily allows uncertain results and thus a precision of less than 100%, the application eventually obtains the accurate result set. Also, the application that receives the result knows at each moment in time whether a result is certain or not, and can thus decide to either use it or not.

As an example, consider an application that displays photos of artists based on the results a certain SPARQL query. After a few HTTP calls, the query client returns 50 matches, all of which have a probability of 99%. The application can decide to already start downloading photos of the 50 matching artists, without displaying them to the user yet. Once 48 of the 50 matches are confirmed, the 48 photos can be displayed immediately; only 2 photos need to be discarded. The user thus sees the photos faster than if they had only been retrieved after full precision was achieved. This example indicates that opportunistic query answering has direct concrete uses in Web applications.

7 Evaluation

In the following, we discuss our evaluation of executing SPARQL queries against TPF interfaces with an AMF feature. From these experiments, we aim to assess whether AMFs are a valuable asset in the metadata dimension. We first describe the experiments and their setup. Then, we discuss their results to validate the three hypotheses of Section 3.2.

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

7.1 Experimental setup

We extended the existing implementations of the TPF client[3] and server[4] to support both Bloom filters and Golomb-coded sets. The server is configured by specifying the AMF and the desired false positive probability. We chose the 32-bit MurMurHash3 hash function for GCS and FNV-1 for the Bloom filter. The server calculates a membership function on the fly for each request for a triple pattern with a single variable.

We ran the experiments with different false positive probabilities p: $1/1024 \approx 0.1\%$, $1/128 \approx 1\%$, and $1/64 \approx 1.6\%$. In each experiment, we executed 250 queries generated from 125 diverse WatDiv SPARQL templates on three interfaces: *i)* regular TPF interface *ii)* TPF with Bloom filters, and *iii)* TPF with GCS. All three cases were tested with both the original and the optimized client; the last two setups were tested with and without opportunistic querying. All experiments were run on a single Amazon EC2 machine with an 8-core Intel Xeon E2680 v2 CPU and 15GB DDR3 RAM, using a query timeout of 3 minutes and the WatDiv 100M triples dataset from [1]. The HTTP requests were routed through an NGINX cache instance to enable HTTP caching and to enforce a realistic Web bandwidth of 1Mbps per request. We published the full result logs online.[5]

7.2 HTTP requests

Tables 1 to 4 summarize the results of the experiments. They compare each AMF-enabled setup against a regular TPF client/server setup, grouping each of the 250 queries on whether they resulted in an *equal*, *lower*, or *higher* measurement for *i)* number of requests, *ii)* time to first result, *iii)* time to 100% recall (i.e., with opportunistic querying enabled), and *iv)* total query execution time. The number of queries per group is indicated, together with their average measurement value in the regular setup, and the average decrease or increase in respectively the *lower* and *higher* groups. For example, the top-left value cell of Table 1 shows that, for Bloom filters with $p = 1/1024$, 126 queries had a lower number of HTTP requests; for each of these 126 queries, the regular setup needed on average 45,213 requests, whereas the AMF-enabled setup required 15,217 fewer requests.

Our experiments show that, with $p = 1/1024$, AMF metadata decreases the number of HTTP calls for roughly half of all considered queries (Bloom: 126 queries or 50.4%; GCS: 123 queries or 49.2%). As expected from the analysis in Section 3, those queries that benefit from improvements are queries with relatively many HTTP requests: the average number of requests per query in the *lower* group is 45,213 (GCS: 45,598), compared to 2,953 (GCS: 2,271) for queries that do not improve. The improvements let us conclude that a substantial number of these 45,000+ requests per query were membership subqueries; the AMF-based query algorithm manages to decrease their number by 15,217 (GCS: 11,761) on average. 43 queries (GCS: 43) result in a slightly higher number of requests, albeit negligible compared to the total number: 10 versus 24,312 (GCS: 18 / 26,919). Note that in general, the number of requests per query is very high because of the potentially high number of results in the WatDiv dataset. While numbers of this scale clearly highlight query patterns, many real-world queries can be evaluated with tighter constraints.

[3] https://github.com/LinkedDataFragments/Client.js/tree/amq

[4] https://github.com/LinkedDataFragments/Server.js/tree/amq

[5] https://github.com/LinkedDataFragments/TPF-Membership-Metadata-Results

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

metric	# requests			1st result time (s)			100% recall time (s)			total time (s)		
queries in group	equal	lower	higher	equal	lower	higher	equal	lower	higher	equal	lower	higher
p=1/1024	81 qrs.	126 qrs.	43 qrs.	177 qrs.	0 qrs.	73 qrs.	152 qrs.	3 qrs.	95 qrs.	154 qrs.	1 qry.	95 qrs.
orig. group avg.	*2,953*	*45,213*	*24,312*	*1*	–	*7*	*96*	*134*	*67*	*96*	*42*	*67*
avg. difference		*−15,217*	*+10*		–	*+6*		*−41*	*+23*		*−32*	*+22*
p=1/128	79 qrs.	134 qrs.	37 qrs.	173 qrs.	0 qrs.	77 qrs.	150 qrs.	3 qrs.	97 qrs.	153 qrs.	1 qry.	96 qrs.
orig. group avg.	*1,469*	*44,712*	*23,623*	*0*	–	*7*	*97*	*134*	*66*	*98*	*42*	*66*
avg. difference		*−14,210*	*+5*		–	*+5*		*−28*	*+24*		*−32*	*+23*
p=1/64	80 qrs.	129 qrs.	41 qrs.	174 qrs.	0 qrs.	76 qrs.	152 qrs.	3 qrs.	95 qrs.	156 qrs.	1 qry.	93 qrs.
orig. group avg.	*2,340*	*44,842*	*24,626*	*1*	–	*7*	*96*	*134*	*66*	*97*	*42*	*66*
avg. difference		*−13,341*	*+15*		–	*+4*		*−41*	*+21*		*−33*	*+21*

Table 1. Comparison of regular TPF versus TPF with Bloom filter setup (greedy TPF algorithm)

metric	# requests			1st result time (s)			100% recall time (s)			total time (s)		
queries in group	equal	lower	higher	equal	lower	higher	equal	lower	higher	equal	lower	higher
p=1/1024	83 qrs.	123 qrs.	44 qrs.	195 qrs.	0 qrs.	55 qrs.	160 qrs.	0 qrs.	90 qrs.	167 qrs.	0 qrs.	83 qrs.
orig. group avg.	*2,271*	*45,598*	*26,919*	*1*	–	*10*	*94*	–	*70*	*91*	–	*72*
avg. difference		*−11,761*	*+18*		–	*+8*		–	*+15*		–	*+16*
p=1/128	83 qrs.	132 qrs.	35 qrs.	196 qrs.	0 qrs.	54 qrs.	154 qrs.	0 qrs.	96 qrs.	153 qrs.	0 qrs.	97 qrs.
orig. group avg.	*2,152*	*45,924*	*21,168*	*1*	–	*11*	*96*	–	*67*	*98*	–	*66*
avg. difference		*−11,594*	*+5*		–	*+8*		–	*+16*		–	*+16*
p=1/64	81 qrs.	122 qrs.	47 qrs.	199 qrs.	0 qrs.	51 qrs.	167 qrs.	2 qrs.	81 qrs.	164 qrs.	2 qrs.	84 qrs.
orig. group avg.	*2,930*	*45,032*	*26,602*	*1*	–	*11*	*91*	*122*	*72*	*93*	*122*	*70*
avg. difference		*−10,521*	*+31*		–	*+7*		*−3*	*+13*		*−3*	*+12*

Table 2. Comparison of regular TPF versus TPF with GCS setup (greedy TPF algorithm)

metric	# requests			1st result time (s)			100% recall time (s)			total time (s)		
queries in group	equal	lower	higher	equal	lower	higher	equal	lower	higher	equal	lower	higher
p=1/1024	82 qrs.	155 qrs.	13 qrs.	166 qrs.	0 qrs.	84 qrs.	200 qrs.	0 qrs.	50 qrs.	173 qrs.	0 qrs.	77 qrs.
orig. group avg.	*1,590*	*18,240*	*11,387*	*1*	–	*5*	*120*	–	*71*	*110*	–	*69*
avg. difference		*–4,920*	*+2*		–	*+5*		–	*+21*		–	*+18*

Table 3. Comparison of regular TPF versus TPF with Bloom filter setup (optimized TPF algorithm)

metric	# requests			1st result time (s)			100% recall time (s)			total time (s)		
queries in group	equal	lower	higher	equal	lower	higher	equal	lower	higher	equal	lower	higher
p=1/1024	87 qrs.	147 qrs.	16 qrs.	199 qrs.	0 qrs.	51 qrs.	203 qrs.	0 qrs.	47 qrs.	209 qrs.	0 qrs.	41 qrs.
orig. group avg.	*2,743*	*18,326*	*10,816*	*1*	–	*9*	*120*	–	*74*	*114*	–	*88*
avg. difference		*−1,154*	*+3*		–	*+5*		–	*+14*		–	*+11*

Table 4. Comparison of regular TPF versus TPF with GCS setup (optimized TPF algorithm)

A similar pattern arises with the optimized TPF algorithm [23], which consumes fewer HTTP requests overall because of full client-side joins, but has potentially longer query times for the same reason. Even more queries benefit from lower request numbers: 155 (62%) for Bloom and 147 (58.8%) for GCS. We see a reduction of roughly the same ratio, both with Bloom filters and GCS, although the absolute request numbers are lower.

The observations generalize to the cases for $p = 1/128$ and $p = 1/64$, albeit with slightly different observations. As is expected from a higher number of false positives, we see a decreasing average gain with increasing p. Interestingly, we see the number of queries with fewer HTTP requests increase slightly for higher p values; we assume this is correlated with the smaller response size, which allows for a higher throughput.

The above results confirm a substantial positive impact on the number of HTTP requests, validating Hypothesis 1.

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

7.3 Query execution time

In all cases (excluding 1 or 2 exceptions), both the first result times and total query times remain the same or even increase, contrarily to what we had expected. As Tables 1 and 2 indicate, about one in three queries have their execution time prolonged with about 20 seconds, or a third of their time. This prolongation is higher for Bloom filters than GCS, which see a more limited effect absolutely (18 seconds) and proportionally (about a quarter). The cause of these elevated query times is likely the increased response size: since the server automatically sends AMFs for all patterns with one variable (even if the client does not use the AMF), the server-side computation time and client-side retrieval time increase. Given a connection of 1Mbps and on-the-fly AMF generation, as in this experiment, the decreased number of requests is apparently insufficient for the considered queries and dataset to result in temporal gains. This is confirmed by the fact that GCS performs better, as GCS representations are encoded more efficiently.

Interestingly, higher false-positive probabilities do not have a profound effect on query time. For the given constrains, the higher number of requests seems to be compensated by the decreased complexity of generating, transferring, and interpreting AMFs. This is an indication that further experimentation with low probabilities might be beneficial.

The prolonged total query time also hinders the effectiveness of opportunistic querying. Whereas its goal is to achieve full recall earlier—at the expense of temporarily allowing <100% precision—the slower overall execution prevented a globally positive result. The potential benefit of opportunistic querying is evidenced by the 3 queries that, with Bloom filters, achieve 100% recall 41 seconds—about a third—earlier. Since opportunistic results have no negative influence on query time, the increased recall times for ± 95 queries must be entirely due to the slower speed of the AMF approach under the 1Mbps and on-the-fly constraints. Should we succeed in speeding up AMF generation and/or transfer time, we could expect to see a broader influence of opportunistic results. Furthermore, the number of false positives that needed to be revoked was either 0 or 1 for all of the considered queries, revealing a low temporary impact on precision.

The obtained results for execution time thus invalidate Hypothesis 2, as we were not able to decrease the time to full recall in general. Further research will need to assess the relation of this observation to on-the-fly generation and bandwidth, and perhaps also even higher false positive rates.

7.4 Server impact

Finally, we measured the average CPU load during query execution for two different AMF configurations and two different false positive probabilities. Compared to the normal server CPU usage (9.2%), the AMF configurations show an increase of 1.6% ($p = 1/1024$), 2% ($p = 1/128$) and 5.7% ($p = 1/64$) for Bloom, and 1% ($p = 1/1024$), 1.6% ($p = 1/128$), and 1.9% ($p = 1/64$) for GCS. This is a very acceptable overhead which does not impact the server's low-cost nature. Bloom has a higher impact than GCS because of the many hashes it needs to calculate, which apparently outweigh the overhead of Golomb compression. Note that all AMF metadata is created at query time and can still benefit from pre-computation and/or caching. Given the limited increase, the aforementioned numbers validate Hypothesis 3.

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

8 Conclusions

The Triple Pattern Fragments API enables client-side SPARQL execution on low-cost servers, at the cost of higher execution time and bandwidth usage. In this paper, we studied the effect of incorporating approximate membership metadata as an interface feature. In particular, we aimed at reducing HTTP requests by avoiding expensive triple membership checks. We observed that, for one third of a set of diverse query types, most of the request overhead are in fact membership subqueries. At the expense of one extra request to fetch the approximate membership metadata, potentially many more could be saved. Indeed, the experimental results confirm a drastic decrease in requests for half of the 250 randomly generated WatDiv queries, while others experience little overhead thanks to local caching. Furthermore, this addition does not affect the low-cost nature of the server, which only has a limited load increase. However, there is a computational overhead on the client for queries that are not improved. An intelligent client should minimize this, by deciding when to use membership metadata based on the query type.

Despite the reduction of requests, the total execution time is higher on average because of long delays introduced to generate AMFs. Therefore, we conclude that this metadata is not suitable for real-time computation. We therefore recommend to pre-compute or pre-cache it in advance. A strong benefit of HTTP caching has been proven for TPF querying [25] due to the limited possible number of requests, and this mechanism can be applied efficiently to TPFs with augmented metadata. While Bloom filters are preferred for lower computation time, the smaller size of Golomb-coded sets would prevail in the presence of caching. To prevent the overhead of generating and transferring AMFs, they could be served in a separate resource that clients explicitly request when needed.

While positive membership tests introduce a slight overhead, this can be compensated by enabling opportunistic querying. Our results show that retracting results after validation is rare and only effects a small number of results. Therefore, it makes sense to design Web applications that can deal with temporarily imprecise results.

A major advantage of adding AMF metadata to the TPF interface is that it happens transparently and in a self-descriptive way. The server can choose freely whether or not to add metadata to a certain response; clients can reactively use metadata when possible, or ignore it when they do not support or need it. Where count metadata has proven crucial for the initial design of TPF querying [25], this first exploration of a new metadata feature was proven an interesting direction. In the future, we could imagine different such types of join optimizations, based on optional selectivity information that servers send as metadata to help clients make intelligent decisions. Studying their impact on real-world scenarios such as human-crafted knowledge bases can shape further directions.

References

1. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: International Semantic Web Conference, pp. 197–212. Springer (2014)
2. Basca, C., Bernstein, A.: Avalanche: Putting the spirit of the Web back into Semantic Web querying. In: Scalable Semantic Web Knowledge Base Systems. pp. 64–79 (2010)
3. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (Jul 1970)

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.

4. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.Y.: SPARQL Web-querying infrastructure: Ready for action? In: 12th International Semantic Web Conference (Nov 2013)
5. Ermilov, I., Martin, M., Lehmann, J., Auer, S.: Linked open data statistics: Collection and exploitation. In: Knowledge Engineering and the Semantic Web, vol. 394, pp. 242–249 (2013)
6. Feigenbaum, L., Williams, G.T., Clark, K.G., Torres, E.: SPARQL 1.1 protocol. Recommendation, W3C (Mar 2013), http://www.w3.org/TR/sparql11-protocol/
7. Filali, I., Bongiovanni, F., Huet, F., Baude, F.: A survey of structured P2P systems for RDF data storage and retrieval. In: Trans. large-scale data-and knowledge-centered systems (2011)
8. Gallager, R., Van Voorhis, D.C.: Optimal source codes for geometrically distributed integer alphabets. Transactions on Information Theory 21(2), 228–230 (Mar 1975)
9. Graefe, G.: Query evaluation techniques for large databases. ACM Computing Surveys 25(2), 73–169 (Jun 1993)
10. Harris, S., Seaborne, A.: SPARQL 1.1 query language. Recommendation, W3C (Mar 2013), http://www.w3.org/TR/sparql11-query/
11. Heine, F.: Scalable P2P based RDF querying. In: Proceedings of the 1st international conference on Scalable information systems (2006)
12. Hose, K., Schenkel, R.: Towards benefit-based RDF source selection for SPARQL queries. Proc. of the 4th International Workshop on Semantic Web Information Management pp. 1–8 (2012)
13. Huang, H., Liu, C.: Estimating selectivity for joined RDF triple patterns. Conference on Information and Knowledge Management pp. 1435–1444 (2011)
14. Li, J., Vuong, S.: Ontsum: A semantic query routing scheme in P2P networks based on concise ontology indexing. In: Advanced Information Networking and Applications (May 2007)
15. Mitzenmacher, M.: Compressed Bloom filters. Transactions on Networking 10(5) (2002)
16. Neumann, T., Weikum, G.: Scalable join processing on very large RDF graphs. In: Proceedings of the International Conference on Management of Data. pp. 627–640. ACM (2009)
17. Oren, E., Guéret, C., Schlobach, S.: Anytime query answering in RDF through evolutionary algorithms. Lecture Notes in Computer Science 5318, 98–113 (2008)
18. Pu, X., Wang, J., Luo, P., Wang, M.: Aweto: Efficient incremental update and querying in RDF storage system. In: Proceedings of the 20th international conference on information and knowledge management. pp. 2445–2448. ACM (2011)
19. Putze, F., Sanders, P., Singler, J.: Cache-, hash-, and space-efficient Bloom filters. Journal of Experimental Algorithmics 14, 4 (2009)
20. Ravindra, P., Hong, S., Kim, H., Anyanwu, K.: Efficient processing of RDF graph pattern matching on MapReduce platforms. In: Proceedings of the 2nd International Workshop on Data Intensive Computing in the Clouds. pp. 13–20 (2011)
21. Rietveld, L., Verborgh, R., Beek, W., Vander Sande, M., Schlobach, S.: Linked data-as-a-service: The semantic web redeployed. In: 12th Extended Semantic Web Conference (2015)
22. Schmachtenberg, M., Bizer, C., Paulheim, H.: Adoption of the linked data best practices in different topical domains. In: International Semantic Web Conference, pp. 245–260 (2014)
23. Van Herwegen, J., Verborgh, R., Mannens, E., Van de Walle, R.: Query execution optimization for clients of Triple Pattern Fragments. In: Extended Semantic Web Conference (Jun 2015)
24. Verborgh, R.: Triple Pattern Fragments. Unofficial draft, Hydra W3C Community Group, http://www.hydra-cg.com/spec/latest/triple-pattern-fragments/
25. Verborgh, R., Hartig, O., De Meester, B., Haesendonck, G., De Vocht, L., Vander Sande, M., Cyganiak, R., Colpaert, P., Mannens, E., Van de Walle, R.: Querying datasets on the Web with high availability. In: 13th International Semantic Web Conference (Oct 2014)
26. Verborgh, R., Mannens, E., Van de Walle, R.: Initial usage analysis of DBpedia's triple pattern fragments. In: Proc. of the 5th Workshop on Usage Analysis and the Web of Data (2015)
27. Zhang, X., Chen, L., Wang, M.: Towards efficient join processing over large RDF graph using MapReduce. In: Scientific and Statistical Database Management. pp. 250–259 (2012)

Pre-print of a paper accepted to the International Semantic Web Conference 2015 (ISWC 2015).
The final publication is available at link.springer.com.